Proceedings of the 2001 IEEE International
Conference on Control Applications
September 5–7, 2001 • México City, México

On Metamodel Composition

Akos Ledeczi, Greg Nordstrom, Gabor Karsai, Peter Volgyesi and Miklos Maroti

Institute for Software Integrated Systems
Vanderbilt University, Nashville TN 37212
http://www.isis.vanderbilt.edu

Abstract– **Computer-based systems (CBS) development integrates various disciplines, such as hardware design, software engineering, and performance modeling, as well as the "base" engineering discipline in which the CBS will operate. As such, use of a "non-native" modeling language is not acceptable when performing CBS design, and rapid specification and development of domain-specific modeling languages (DSMLs) is necessary. We advocate a UML -based metamodeling technique to DSML specification and generation. A key feature of our approach is the composition of new metamodels from existing metamodels through the use of three newly defined UML operators—*equivalence*, *implementation inheritance*, and *interface inheritance*. This paper describes the development of these new operators, details how they are used in metamodel composition, and presents examples of metamodel composition.**

Index terms–**metamodeling, model composition, model-based computing, UML**

I. Introduction

Computer-based systems (CBSs), where functional, performance, and reliability requirements demand the tight integration of physical processes and information processing, are among the most significant technological developments of the past 20 years [1]. Their development integrates various disciplines, such as hardware design, software engineering, and performance modeling, as well as the "base" engineering discipline in which the CBS will operate. In designing CBS hardware and software, one must use *domain-specific* terminology, concepts and techniques, and modeling such CBS systems requires a domain-specific modeling language. Modeling languages designed to capture interesting properties of software systems (e.g. UML [5]) are rarely suitable for modeling an entire CBS, because the "entire system" includes not only the software, but the hardware and the CBS operating environment as well. Also, while there are some aspects of UML that make it suitable for modeling dynamic, reactive systems (e.g. state charts), it is inadequate for capturing models in the form of, for instance, Laplace transforms or differential equations.

Because mature engineering disciplines (e.g. control theory or chemical engineering) have their own languages, requiring designers to use a "non-native" modeling language is not acceptable. Additionally, in CBS design projects we often need to integrate models across engineering disciplines. Both of these goals can be achieved by using appropriate tools, but at a very high cost—the development of customized modeling and integration solutions is very expensive. Our approach is to use a higher-level, meta-level modeling language [2][3][4]. The meta-language is not used for defining domain *models*, but rather for defining domain-modeling *languages*. The various components of such a language—its ontology, syntax, and interpretation (i.e. semantics)—are specified formally in a metamodel, and the actual domain-specific modeling language (DSML) is synthesized automatically from the metamodel.

While UML and OCL [6] may not be suitable for modeling within most engineering domains, they represent a good choice for constructing metamodels. UML class diagrams are used to specify modeling entities and relationships, and OCL is used to specify the static semantics of the DSML. (Strictly speaking, some of the static semantics are specified using the associations among the class objects, but in general, class diagrams alone are not sufficiently expressive to fully define the static semantics of a DSML). The UML diagrams play a similar role to that of BNF (Backus-Naur Form): they specify the grammar of a language, that can generate all legal "sentences" (i.e. models). Using UML and OCL to construct metamodels has been discussed at length elsewhere [2][3]. What has not been addressed until now is the composition of metamodels from existing metamodels. This paper describes our initial investigations into such compositions.

II. Composition of Metamodels

Metamodel composition is necessary for several reasons. A metamodel is the embodiment of a modeling paradigm—the set of axioms, notions, idioms, abstractions, and techniques that govern how systems within the domain are to be modeled. As such, metamodels represent a large investment

0-7803-6733-2/01/$10.00 © 2001 IEEE

in the understanding of a particular engineering domain. When defining a new modeling language, often some of the concepts that appear in an existing language will also appear in the new language. When defining the new language, we would like to call upon this previously documented domain knowledge when constructing the new metamodel—i.e. we want to be able to reuse all or part of existing metamodels when building new metamodels. In a similar manner, we can compose specific metamodels from abstract metamodels (i.e. metamodels that do not represent a DSML *per se*, but capture some general modeling behavior such as hierarchy or inheritance). Such an approach is only possible if, as part of the composition process, domain-specific concepts and constraints can be added to the resultant metamodel. An example might be the composition of a signal flow metamodel with a generalized metamodel of inheritance, resulting in a hierarchical signal flow modeling language. Such a modular, compositional approach to metamodel specification and construction promises to reduce development costs and risk, while simultaneously increasing the quality and functionality of the metamodel.

When considering metamodel construction via composition, several issues and questions arise. First and foremost is the overall applicability of combining two existing modeling languages into a new DSML. Just because two languages *can* be combined, does it make sense to actually combine them? Also, how do the individual concepts contained in each language compliment each other? Which language elements of the source metamodels survive the composition and which do not? How are the individual constraints embedded in each metamodel combined? Finally, what becomes of the models that were created using the original languages? Can they be understood (i.e. edited) by the newly formed DSML? We attempt to answer these and other questions in this paper.

A. Extending UML for metamodel composition

As discussed in [3], metamodels themselves are created using a modeling language specifically designed for specifying modeling languages, properly referred to as a metamodeling language. This metamodeling language is also specified using a metamodel, known as a meta-metamodel. As mentioned earlier, this metamodeling language is currently based on UML and OCL.

The primary design goal of the composable metamodeling environment is to leave the original metamodels intact, still able to be used independently from any composition they may be a part of. This ensures that models created using DSMLs derived from the original metamodels are still valid—the fact that their metamodel also participates in a composition does not affect a model's ability to function exactly as it did before the composition. Second, the newly composed metamodel defines a DSML that is capable of editing models created using the original DSML. Finally, such an approach to composition makes the creation of libraries of metamodels possible.

Limitations of UML made it necessary to develop three new UML operators for use in combining metamodels together: an *equivalence* operator, an *implementation inheritance* operator, and an *interface inheritance* operator. The semantics and use of each of these constructs are discussed below.

1) Equivalence operator

The equivalence operator is used to show a full union between two UML class objects. The two classes cease to be two separate classes, but instead form a single class. Thus, the union includes all attributes and associations, including generalization, specialization, and containment, of each individual class. Equivalence can be thought of a defining the "join points" or "composition points" of two or more source metamodels, and has the effect of adding the attributes and associations of one class to those of another class (possibly in a different class diagram). Before such equivalence operations can be interpreted or used as the basis for model translation, they must be reduced to a more basic set of UML associations and constraints. This reduction represents a "flattening" of more complex UML diagrams into "pure" UML representations. Gogolla and Richters have investigated UML diagram equivalence and developed methods for performing such reductions [7]. We are currently investigating their techniques for possible inclusion in the metamodel interpretation phase of our research.

2) Implementation inheritance operator

The semantics of UML specialization (i.e. inheritance) is straightforward: specialized (i.e. child) classes contain all the attributes of the general (parent) class, and can participate in any association the parent can participate in. However, during metamodel composition, there are cases where finer-grained control over the inheritance operation is necessary. Therefore, we have introduced two new types of inheritance operations between class objects—*implementation inheritance* and *interface inheritance*.

In implementation inheritance, the child inherits all of the parent's attributes, but only the containment associations where the parent functions as the container. No other associations are inherited. Implementation inheritance is represented graphically by a UML generalization icon containing a solid black dot. This can be seen in the left hand diagram of Figure 1 below, where implementation inheritance is used to derive class *X1* from class *B1*. In this case, *X1* inherits the age attribute from *B1*, as well as the association allowing objects of type *C1* to be contained in objects of type *B1*. In other words, *X1*-type objects can contain *C1*-type objects. Because *B1*-type objects can contain other *B1*-type objects, *X1*-type objects can contain

objects of type *B1* **but not** *X1*. Note that *D1*-type objects can contain objects of type *B1* but not objects of type *X1*.

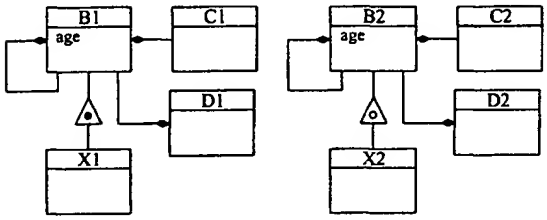

Figure 1. Implementation- (left) and Interface (right) inheritance

3) Interface inheritance operator

The right side of Figure 1 shows interface inheritance between *B2* and *X2* (the unfilled circle inside the inheritance icon denotes interface inheritance). Interface inheritance allows no attribute inheritance but does allow full association inheritance, with one exception: containment associations where the parent functions as the container are not inherited. Therefore, in this example, *X2*-type objects can be contained in objects of type *D2* and *B2*, but no objects can be contained in *X2*-type objects, not even other X2-type objects. Note that the age attribute is not inherited by *X2*.

The union of implementation- and interface inheritance is the normal UML inheritance, and their intersection is null. It should also be noted that these operators could have been implemented using UML associations with stereotypes, but we selected a graphical representation, for two reasons. First, our application area (GME2000, discussed briefly below) is graphical in nature, and we believe graphical operators are more readily interpreted by humans and are intrinsically more readable, especially when the semantics are fixed and well known *a priori*, as these are. Second, interface- and implementation inheritance are semantically much closer to regular inheritance than to associations. Therefore, the use of association with stereotypes would be somewhat misleading.

B. Composing metamodels via implementation- and interface inheritance

Figure 2 shows the composition of two metamodels to form a third, new metamodel. The goal in this example is to combine a signal flow modeling paradigm (**Signal Flow**) with a finite state machine (**FSM**) modeling paradigm to form a new modeling paradigm.

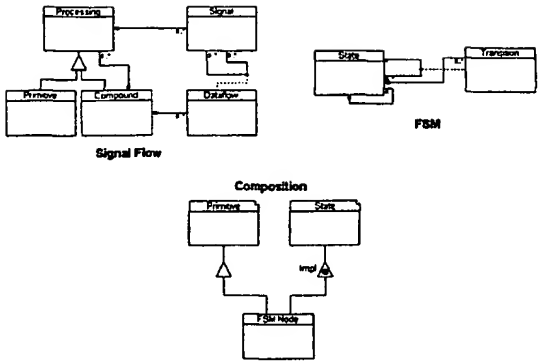

Figure 2. Metamodel composition

In the new metamodel we want to allow leaf nodes of the signal flow graph (Primitives) to have hierarchical state machine implementation. In other words, we want a specialized Primitive that also inherits the implementation of the State class object. However, we must not allow Primitives to "act" as States, i.e. we must not permit States to contain Primitives (hence the use of implementation inheritance between the State and FSM Node classes).

Note the Primitive and State objects that appear in the Composition metamodel. These are not UML class objects, but are *proxy* objects (proxy objects are denoted here as class objects with a "folded" upper right corner). Proxy objects refer to existing UML class objects. The use of proxy objects decouples the original metamodels from the composed metamodel. In this way, proxies can be included in new metamodels as necessary, while the source metamodels remain unchanged.

It is important to observe that the use the equivalence, implementation- and interface inheritance operators just discussed are a notational convenience, and in no way change the underlying semantics of UML. In fact, each operator has an equivalent "pure" UML representation, and as such, each composed metamodel could be represented without the new operators. However, such metamodels are generally significantly more "cluttered," making the diagrams more difficult to read and understand.

III. METAMODELING ENVIRONMENT

An interesting aspect of the GME 2000 tool suite is that the same set of tools is used for metamodeling as for domain modeling. The metamodeling problem can be thought of as just another domain, the field of domain-specific design environments. Hence, the metamodeling language is just another domain language. The meta-specifications that configure GME 2000 are generated by the metamodeling translator from the metamodels. The metamodeling environment itself is generated by the same translator when translating the meta-metamodels.

The metamodeling language in GME 2000 is the UML class diagram notation [3]. The metamodels fully specify the domain modeling language, or more precisely, its concrete syntax. They do not, at least not entirely, specify the static semantics of the language. By static semantics we mean the set of rules that specify the well-formedness of domain models. UML class diagrams do allow the specification of some basic rules, for example, the multiplicity of associations. For more complex semantic specifications, however, UML includes the Object Constraint Language (OCL) [4], a textual predicate logic language. GME 2000 adopts OCL as well; metamodels consist of UML class diagrams and OCL constraints.

A. Metamodel composition in GME 2000

Just as the reusability of domain models from application to application is essential, the reusability of metamodels from domain to domain is also important. Ideally, a library of metamodels of important sub-domains should be made available to the metamodeler, who can extend and compose them together to specify domain languages. These sub-domains might include different variations of signal-flow, finite state machines, data type specifications, fault propagation graphs, Petri-nets, etc. The extension and composition mechanisms must not modify the original metamodels, just as subclasses do not modify base classes in OO programming. This way, changes in the metamodel libraries, which reflect a better understanding of the given domain, can propagate automatically to the metamodels that utilize them. Furthermore, by precisely specifying the extension and composition rules, models specified in the original domain language can be automatically translated to comply with the new extended and composed modeling language.

Consider the left hand side of Figure 3 with the two UML classes *Base* and *Sub*. Sub is derived from Base through both implementation inheritance (denoted by a filled circle inside a triangle) and interface inheritance (denoted by an empty circle inside a triangle). By applying the interface inheritance operator, we get the equivalent class diagram consisting of *Base2a* and *Sub2a*. Similarly, applying implementation inheritance first, we get *Base2b* and *Sub2b*. Finally, continuing from either one and applying the remaining inheritance operator, we end up with the class diagram of *Base3* and *Sub3*. Notice that this matches exactly the diagram we would get by applying regular UML inheritance to Base and Sub instead of the two new operators.

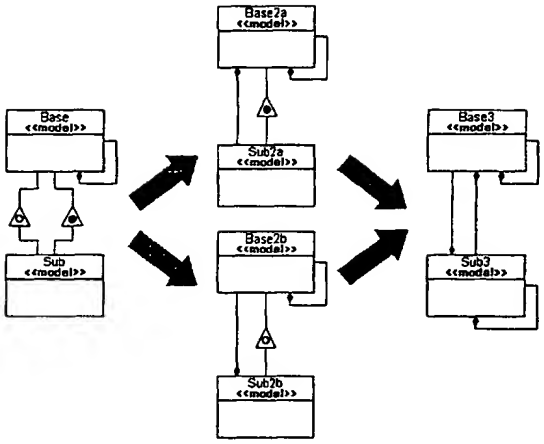

Figure 3 Interface and implementation inheritance

IV. ILLUSTRATIVE EXAMPLE

It is probably best to illustrate these ideas through an example. Figure 4 shows three metamodels. The first (*SignalFlow*) specifies a hierarchical signal flow modeling language. *Processing* is an abstract base class. *Compound* is a composite model that can contain other Compounds and *Primitives*. Primitives are the leaf nodes that implement the elementary computation in the graph (they may have an implementation associated with them in a traditional programming language, for example.) The signal flow connections are implemented by connecting *InputSignals* and *OutputSignals* together with *Dataflow* connections. The second metamodel (*FSM*) describes a simple hierarchical finite state machine paradigm. *States* can contain other States that can be connected together by *Transition* connections.

We assume that these metamodels were already in existence, and were imported from a metamodel library. We seek to combine them together according to the following rules. We would like to have a new kind of Primitive (*FSMNode*) that can contain a finite state machine specifying its implementation. However, we do not want a State to be able to contain this new kind of model. Furthermore, we want to make selected InputSignals and OutputSignals of any FSMNode to be mapped to certain States it contains using connections. (This could mean, for example, that the data values associated with those signals are accessible from the implementation associated with the given State.)

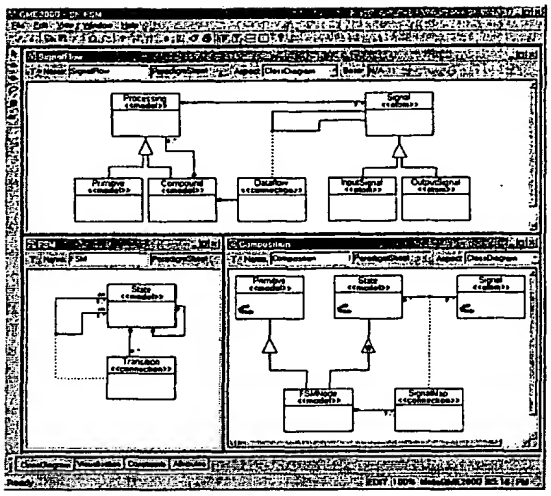

Figure 4 Composed Signal Flow and FSM Metamodels

These rules are accomplished by the third metamodel (*Composition*). The new FSMNode class inherits from both Primitive and State. Notice that the curved arrow inside these classes indicate that they are proxies to existing UML classes defined elsewhere. Inheriting from State through the standard UML inheritance would mean that a State could contain an FSMNode violating one of our rules. Instead, we use implementation inheritance to accomplish exactly what we want: an FSMNode that can contain whatever a State can, but that cannot act as a State; i.e. it cannot be inserted into a State. (Neither can FSMNodes connected together by Transitions.) Notice how the new SignalMap connection connecting States and Signals is also introduced in the Composition metamodel.

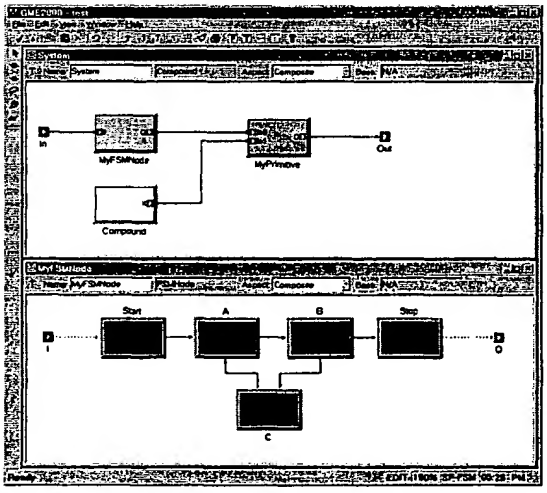

Figure 5 Combined Signal Flow and FSM Models

Figure 5 shows a simple model in the target environment. The System model contains a Compound, a Primitive and an FSMNode. The bottom window shows the contents of the latter: a simple state machine with Signals mapped to certain States.

V. Conclusions

We introduced the Generic Modeling Environment (GME 2000), a configurable domain-specific design environment. Other similar environments include Dome by Honeywell Research [8] and MetaEdit+ by MetaCASE Consulting. [9] presents a brief comparison of these three environments. One of the unique features of GME 2000 is its UML class diagram-based metamodeling environment. Our experience showed that some extensions to this standard notation were necessary, primarily to support metamodel composition. The composable metamodeling environment is a brand new addition to GME 2000. We are in the process of applying it to several real world application domains. An early indication of its usability is the significantly increased readability of its own meta-metamodels.

VI. References

[1] J. Sztipanovits, "Engineering of Computer-Based Systems: An Emerging Discipline," *Proceedings of the IEEE ECBS'98 Conference*, 1998.

[2] A. Ledeczi, et al., "Metaprogrammable Toolkit for Model-Integrated Computing," Proceedings of the IEEE ECBS'99 Conference, 1999.

[3] Nordstrom G.: "Metamodeling - Rapid Design and Evolution of Domain-Specific Modeling Environments", Ph.D. Dissertation, Vanderbilt University, 1999.

[4] G. Karsai, et al., "Specifying Graphical Modeling Systems Using Constraint-based Metamodels," IEEE Intl. Symp. On CACSD, Anchorage, Alaska, Sep. 2000.

[5] UML Semantics, ver. 1.1, Rational Software Corporation, et al., September 1997.

[6] Object Constraint Language Specification, ver. 1.1, Rational Software Corporation, et al., Sept. 1997.

[7] Gogolla, M., and Richters, M. "Equivalence rules for UML class diagrams," *Proceeding of the UML'98 Workshop*, P.-A. Muller and J. Bezivin, Eds., Universite de Haute-Alsace, Mulhouse, pp. 86-97, 1998.

[8] Dome Official Web Site, Honeywell, 2000, http://www.src.honeywell.com/dome/

[9] MetaEdit+ Official Website, MetaCase Consulting, http://www.metacase.com